Feedback Loop Network Analyzer

Shreeharshini Dharanesh Murthy, Qiang Du

January 26, 2023

1 Introduction

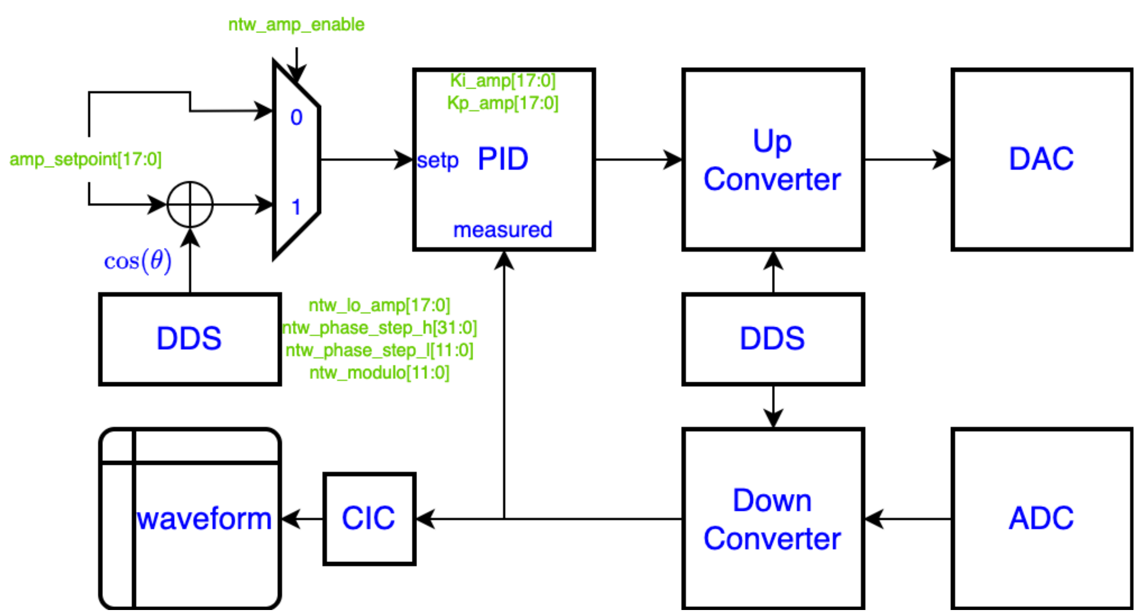
LLRF system feedback loops have optimization routine, to determine optimal operating point, using a 'network analyzer' probe technique for feedback frequency response analysis, the well known Bode plot. This is done by injecting a known excitation tone to the feedback controller through the setpoint, and measure the response amplitude and phase. If one can scan excitation tone over the bandwidth of interest, the Bode plot can be obtained. The same process is applied to both amplitude and phase loops. In this lab you have the choice of either amplitude or phase loop characterization.

1.1 Firmware details

The excitation tone is generated by an additional DDS unit in the firmware, through the usual registers of phase step high, low and modulo, plus its amplitude control register.

The excitation tone would only be added to the normal loop setpoint if the network analyzer is enabled, through register `ntw_amp_enable`.

Firmware level block diagram for amplitude loop:



The same is for phase loop, except the register names has `phs` instead of `amp`.

1.2 Steps

1. Confirm system is in normal closed-loop operation state, with certain conditions such as RF power level.
2. A known excitation signal (a single frequency tone, or a chirp) is generated and added on the setpoint of either amplitude or phase loop. This signal is synchronous to waveform triggering, so the measured result would be consistent across waveforms.
3. Record waveform of interest, for example, cavity cell amplitude or phase waveform.
4. If using a single frequency tone excitation, scan the tone frequency across analyze bandwidth of 10 Hz to 100 kHz, and record waveform accordingly.
5. Compute frequency response (amplitude and phase) between the excitation signal and measured waveform, derive bode plot.
6. Explore different gain settings, find amplitude margin and phase margin for stability.

1.3 EPICS PVs:

Note: all raw registers like `reg_*` have separate write and read-back PVs.

To write a raw register such as `USPAS:LLRF:reg_amp_loop_enable`:

```
caput USPAS:LLRF:reg_amp_loop_enable 0
```

To read it back, read the PV with `_RBV` as suffix.

```
caget USPAS:LLRF:reg_amp_loop_enable_RBV
```

Waveform:

- **`USPAS:LLRF:CavCel:AWF`** Holds cavity cell amplitude data, in counts
- **`USPAS:LLRF:CavCel:PWF`** Holds cavity cell phase data, in degrees
- **`USPAS:LLRF:CavCel:TWF`** Holds cavity cell time data, in microsecond
- **`USPAS:LLRF:ACQ_SAMP_PERIOD`** Waveform sampling period
- **`USPAS:LLRF:ACQ_DECIM`** Waveform decimation value

Amplitude loop:

- **`USPAS:LLRF:Loop:AmpSetp`** Holds amplitude loop setpoint value, calibrated with waveform
- **`USPAS:LLRF:reg_amp_setpoint`** Holds amplitude loop setpoint value, raw
- **`USPAS:LLRF:reg_amp_loop_enable`** Enables amplitude loop
- **`USPAS:LLRF:reg_amp_loop_reset`** Resets amplitude loop
- **`USPAS:LLRF:reg_Kp_amp`** Amplitude loop proportional gain
- **`USPAS:LLRF:reg_Ki_amp`** Amplitude loop integral gain

Phase loop:

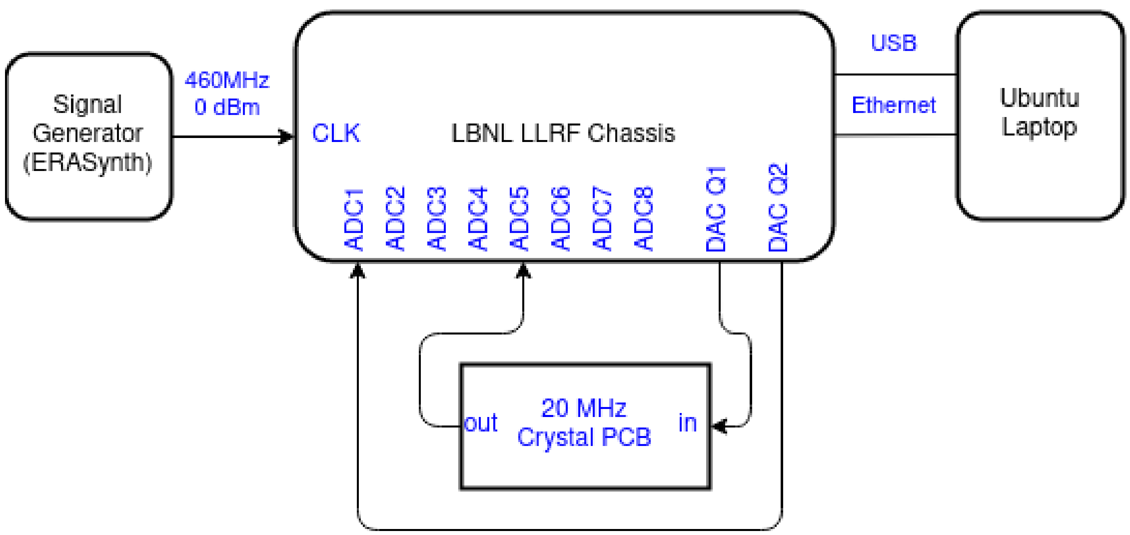
- **`USPAS:LLRF:Loop:PhsSetp`** Holds phase loop setpoint value, calibrated with waveform
- **`USPAS:LLRF:reg_phs_setpoint`** Holds phase loop setpoint value, raw

- **USPAS:LLRF:reg_phs_loop_enable** Enables phase loop
- **USPAS:LLRF:reg_phs_loop_reset** Resets phase loop
- **USPAS:LLRF:reg_Kp_phs** Phase loop proportional gain
- **USPAS:LLRF:reg_Ki_phs** Phase loop integral gain

Network analyzer:

- **USPAS:LLRF:reg_ntw_amp_enable** Enables amplitude loop network analzyer feature
- **USPAS:LLRF:reg_ntw_phs_enable** Enables phase loop network analzyer feature
- **USPAS:LLRF:reg_ntw_lo_amp** Network analzyer amplitude value
- **USPAS:LLRF:reg_ntw_phase_step_h** Network analyze major resolution value, like DDS minor resolution
- **USPAS:LLRF:reg_ntw_phase_step_l** Network analyze minor resolution value, like DDS major resolution
- **USPAS:LLRF:reg_ntw_modulo** Network analyze modulo value, like DDS modulo

1.4 Hardware setup



Connect the system as shown in the diagram, so we have both direct loop back on `ADC1` and with 20 MHz crystal in the system on `ADC5`. The firmware has identical `DAC Q1` and `DAC Q2` output, as we previously measured on the DAC characterization lab.

1.5 Firmware and software setup

- Configure FPGA chassis using the provided `marble_zest_top_uspas.bit`;
- Start EPICS IOC on the connected laptop computer;
- Run Phoebus GUI on the connected laptop computer;

- Note: Phase shift/error caused by cic decimation in the FPGA waveform capture module is 14.442 radians. Measured by sweeping the decimation factor from 1 to 127 at a given frequency.

2 Exercises

2.1 Run EPICS Phoebus interface

2.2 Close both amplitude and phase loops

2.3 Enable either amplitude or phase network analyzer feature and run frequency sweep routine

2.4 Capture cavity cell amplitude/phase data along with the time data. Compensate for CIC phase error.

2.5 Calculate closed loop transfer function

2.6 Calculate open loop transfer function based on closed loop transfer function

2.7 Plot amplitude and phase bode plots

2.8 Calculate gain and phase margins

[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import time
import numpy.fft
import ntw_analyzer
from scipy import signal
from numpy import savetxt, loadtxt
from scipy import signal
from scipy.signal.windows import flattop

plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['axes.grid'] = True
plt.rcParams['axes.grid.which'] = "both"
plt.rcParams['grid.linewidth'] = 0.5
plt.rcParams['grid.alpha'] = 0.5
plt.rcParams['font.size'] = 8
```

[12]:
```python
from epics import PV
import os
os.environ['EPICS_CA_ADDR_LIST'] = 'localhost'
os.environ['EPICS_CA_AUTO_ADDR_LIST'] = 'NO'

prefix = 'USPAS:LLRF:'
pvs = {
    'cav_cel_amp_wfm': PV(prefix + 'CavCel:AWF'),
```

```
    'cav_cel_phs_wfm': PV(prefix + 'CavCel:PWF'),
    'cav_cel_time_wf': PV(prefix + 'CavCel:TWF'),

    'amp_loop_setp': PV(prefix + 'Loop:AmpSetp'),
    'amp_loop_setp_eslo': PV(prefix + 'Loop:AmpSetp.ESLO'),
    'amp_loop_kp': PV(prefix + 'reg_Kp_amp'),
    'amp_loop_ki': PV(prefix + 'reg_Ki_amp'),
    'phs_loop_setp': PV(prefix + 'Loop:PhsSetp'),
    'phs_loop_kp': PV(prefix + 'reg_Kp_phs'),
    'phs_loop_ki': PV(prefix + 'reg_Ki_phs'),
    'amp_loop_enable': PV(prefix + 'reg_amp_loop_enable'),
    'phs_loop_enable': PV(prefix + 'reg_phs_loop_enable'),

    'ntw_amp_enable': PV(prefix + 'reg_ntw_amp_enable'),
    'ntw_phs_enable': PV(prefix + 'reg_ntw_phs_enable'),
    'ntw_reset': PV(prefix + 'reg_ntw_reset'),
    'ntw_lo_amp': PV(prefix + 'reg_ntw_lo_amp'),
    'ntw_phase_step_h': PV(prefix + 'reg_ntw_phase_step_h'),
    'ntw_phase_step_l': PV(prefix + 'reg_ntw_phase_step_l'),
    'ntw_modulo': PV(prefix + 'reg_ntw_modulo'),

    'amp_loop_reset': PV(prefix + 'reg_amp_loop_reset'),
    'phs_loop_reset': PV(prefix + 'reg_phs_loop_reset'),
    'samp_period': PV(prefix + 'ACQ_SAMP_PERIOD'),
    'wave_decim': PV(prefix + 'ACQ_DECIM'),
    'wave_samp_per': PV(prefix + 'reg_wave_samp_per')
}
```

[4]:
```
import sys
sys.path.append('..')
from dds.dds import reg2freq, calc_dds
```

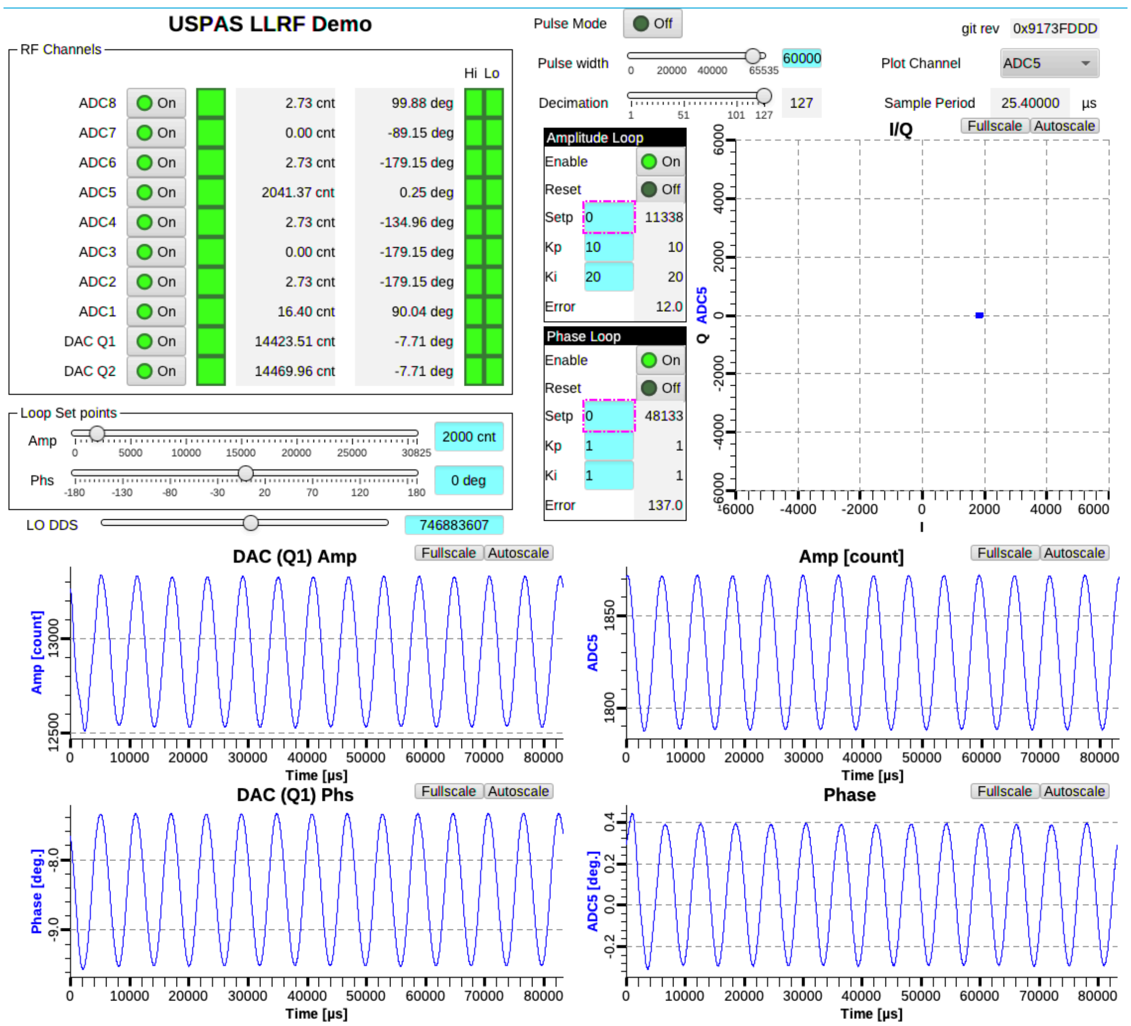
[9]:
```
f_tone = 100  # Hz
fs = 115e6
den = 23
num = f_tone * den / fs
ph, pl, modulo = calc_dds(num, den, dwh=32, dwl=12)
print(ph, pl, modulo)
fdds = reg2freq(ph, pl, modulo, fs, dwh=32, dwl=12)
print(f'New DDS Freq: {fdds:.2f} Hz')
```

```
3734 3087 2
major resolution:  0.027 Hz
minor resolution:  0.000 Hz
modulo resolution: 0.000 Hz
New DDS Freq: 100.00 Hz
```

```
[10]: pvs['ntw_lo_amp'].value = 150
pvs['ntw_modulo'].value = modulo
pvs['ntw_phase_step_l'].value = pl
pvs['ntw_phase_step_h'].value = ph
```

Turn on amplitude ntw excitation

```
[13]: pvs['ntw_amp_enable'].value = 1
```




```
[14]: def drive_ntw(f_tone=100, den=23, fs=115e6):
    num = f_tone * den / fs
    ph, pl, modulo = calc_dds(num, den, dwh=32, dwl=12)
    pvs['ntw_lo_amp'].value = 150
    pvs['ntw_modulo'].value = modulo
    pvs['ntw_phase_step_l'].value = pl
    pvs['ntw_phase_step_h'].value = ph
```

[15]:
```
awf = pvs['cav_cel_amp_wfm'].get()
```

[16]:
```
awfs = []
for f in range(0, 1000, 10):
    # print(f'Driving f_tone: {f} Hz')
    drive_ntw(f_tone=f)
    time.sleep(0.5)
    awfs.append(pvs['cav_cel_amp_wfm'].get())
```

[30]:
```
arr = np.array(awfs)
np.save('amp100.npy', arr)
```